Fault-Tolerant Compact Routing Schemes for General Graphs

Shiri Chechik

Department of Computer Science and Applied Mathematics, The Weizmann Institute of Science, Rehovot, Israel.

Abstract

This paper considers compact fault-tolerant routing schemes for weighted general graphs, namely, routing schemes that avoid a set of failed (or *forbidden*) edges. We present a compact routing scheme capable of handling multiple edge failures. Assume a source node s contains a message M designated to a destination target t and assume a set F of edges crashes (unknown to s). Our scheme routes the message to t (provided that s and t are still connected in $G \setminus F$) over a path whose length is proportional to the distance between s and t in $G \setminus F$, to $|F|^3$ and to some poly-log factor. The routing table required at a node v is of size proportional to the degree of v in G and some poly-log factor. This improves on the previously known fault-tolerant compact routing scheme for general graphs, which was capable of overcoming at most 2 edge failures.

Keywords: forbidden sets, fault-tolerance, compact routing

1. Introduction

Routing is one of the most fundamental problems in distributed networks. A routing scheme is a mechanism that allows delivering a message from a source node s to a target node t. In the routing process, each node in the network may receive packets of information and has to decide if the message already reached its destination or not. If the message did not reach its destination then the node has to forward the message to one of its neighbors, using only its local information and the header of the message, containing the label of the destination and possibly some other data. A key concern in designing a routing scheme is to minimize the worst case multiplicative stretch, namely, the maximum ratio between the length of a path obtained by the routing scheme and the length of the shortest path between the source and the destination. Another important goal is to minimize the size of the routing tables stored at the nodes. Subsequently, the focus in designing a routing scheme is often on the

Email address: shiri.chechik@weizmann.ac.il (Shiri Chechik)

tradeoff between the size of the routing tables and the maximum stretch of the resulting routes.

The problem of designing compact and low stretch routing schemes has been extensively studied (e.g. [17, 3, 5, 16, 9, 12, 15]). The first tradeoff between the size of the routing tables and the maximum multiplicative stretch of the routing scheme was considered by Peleg and Upfal [17]. In this paper, the total size of the routing tables was considered (as opposed to the maximum table size of the nodes) and only unweighted graphs were considered. Later, weighted graphs were considered by Awerbuch et al. [3] and a bound on the maximum table size was achieved. The weights on the edges correspond to distances, namely, the distance of a path is the sum of the weights of its edges. Awerbuch et al. show how to construct a routing scheme with maximum table size of $\tilde{O}(n^{1/k})$ and with multiplicative stretch dependent on k ($O(k^2 9^k)$) for any integer $k \geq 1$. Further improvements were later introduced by Awerbuch and Peleg [5] for general integer $k > 1$, by Cowen [9] for $k = 3$ and by Eilam et al. [12] for $k = 5$. These tradeoffs were later improved by Thorup and Zwick [19], obtaining the best one known so far. They present a routing scheme with table size of $\tilde{O}(n^{1/k})$ and a multiplicative stretch of $2k - 1$ using handshaking by which the source and target agree on an $o(\log^2 n)$ bit header that is attached to all packets, or a stretch of $4k - 5$ without using handshaking. Corresponding lower bounds were presented in [17, 13, 14, 20].

In this paper, we consider a natural and significant extension of routing schemes for general weighted graphs, that may better suit many network settings. Suppose that some of the links crash from time to time and it is still required to deliver messages between the nodes, if possible, without recomputing the routing tables and the labels. Given a set F of edge failures, the multiplicative stretch is with respect to the distance between the source node and the destination node in the surviving graph $G \setminus F$. The objective is once again to minimize both the routing table sizes and the stretch. This extension was suggested in [8, 21], which consider this problem for graphs of bounded treewidth or cliquewidth. It is shown how to assign each node a label of size $O(\log^2 n)$ (with some dependency on the tree/clique width) such that given the labels of the source and target and the labels of a set F of "forbidden" vertices, the scheme can route from the source to the target on the shortest path in $G \setminus F$.

Later, in [1], the same extension was considered for unweighted graphs of bounded doubling dimension. It is shown how to construct a labeling scheme for a given unweighted graph of doubling dimension α such that for any desired precision parameter $\epsilon > 0$, the labeling scheme stores an $O(1 + \epsilon^{-1})^{2\alpha} \log^2 n$-bit label at each vertex. Given the labels of two end-vertices s and t, and the labels of a set F of "forbidden" vertices, the scheme can route on a path of length at most $1 + \epsilon$ times the distance from s to t in $G \setminus F$.

Note that in both [8] and [1] the assumption is that the labels of the faulty nodes are known to the source.

For weighted general graphs, the design of fault-tolerant compact routing schemes was considered in [6]. However, the scheme of [6] only dealt with up to 2 edge failures and bounded only the total size of the routing tables at all

the nodes. It is shown how to construct a routing scheme for a given parameter k, that in the presence of a forbidden edge set F of size at most 2 (unknown to the source), routes the message from the source s to the destination t over a path of length $O(k \cdot \mathbf{dist}(s,t,G \setminus F))$, where $\mathbf{dist}(s,t,G \setminus F)$ is the distance from s to t in $G \setminus F$. The total amount of information stored in the vertices of G is $O(kn^{1+1/k} \log(nW) \log n)$, where W is the weight of the heaviest edge in the graph (assuming the minimal weight is 1).

Our contributions. In this paper we present a compact fault-tolerant routing scheme for weighted undirected general graphs. We manage to design a compact routing scheme that handles multiple edge failures. More specifically, we prove the following theorem. [1]

Theorem 1.1. *Given a graph $G = (V, E)$ with edge weights ω such that $\omega(e) \in [1, W]$ for every edge e and a parameter k, one can efficiently construct a routing scheme that given a source node s and a target node t, in the presence of a set of failures F (unknown to s), can route a message from s to t in a distributed manner over a path of length at most $O(|F|^2 \cdot (|F| + \log^2 n) \cdot k \cdot \mathbf{dist}(s,t,G \setminus F))$. The scheme requires assigning to each node a label of length $O(\lceil \log(nW) \rceil \cdot \log n)$ bits and the routing table of a node v is of size at most $O(\lceil \log(nW) \rceil \cdot k \cdot n^{1/k} \cdot \deg(v) \cdot \log^2 n)$ bits. The message passed during the routing process is of size $O(|F| \cdot \log n)$ bits.*

2. General Framework

In this section we outline the general structure of our routing scheme. Let $G(V, E)$ be an n-node undirected weighted graph with edge weights ω such that $\omega(e) \in [1, W]$ for every edge e (hence nW is an upper bound on the diameter). For a given graph H and a tree T such that $V(T) \subseteq V(H)$, let $H|_T$ be the subgraph of H induced on the vertices of T.

Our results are based on the well known construction of tree cover, defined as follows.

Tree covers:. Let $G(V, E)$ be an undirected graph with edge weights ω, and let ρ, k be two integers. Let $B_\rho(v) = \{u \in V \mid \mathbf{dist}(u, v, G) \leq \rho\}$ be the ball of vertices of (weighted) distance at most ρ from v. A *tree cover* $\mathbf{TC}(G, \omega, \rho, k)$ is a collection $\mathcal{T} = \{T_1, \ldots, T_\ell\}$ of rooted trees in G, with $V(T) \subseteq V$ and a root $r(T)$ for every $T \in \mathcal{T}$, with the following properties:

(i) For every $v \in V$ there exists a tree $T \in \mathcal{T}$ such that $B_\rho(v) \subseteq T$.

(ii) For every $T \in \mathcal{T}$ and every $v \in T$, $\mathbf{dist}(v, r(T), T) \leq (2k-1) \cdot \rho$.

(iii) For every $v \in V$, the number of trees in $\mathcal{T}$ that contain v is $O(k \cdot n^{1/k})$.

[1] Notice that by setting $k = \log n$, the term $n^{1/k}$ in the theorem becomes a constant.

Proposition 2.1 ([4, 7, 16]). *For any ρ and k, there exists a tree cover $\mathbf{TC}(G, \omega, \rho, k)$ constructible in time $\tilde{O}(mn^{1/k})$.*

A basic building block of our routing scheme is a procedure for routing on subtrees of the graph. For that purpose we use the standard interval routing scheme on trees ([18, 22]) defined as follows. Traverse the tree T according to some Euler tour, let $L(T)$ be the list of the vertices of T in the order in which they were encountered during this Euler tour, keeping only the first occurrence of each vertex. Each node v is identified with its index $\ell(v)$ in $L(T)$. For each node v let f_v be the descendant of v with the largest $\ell(f_v)$. It's not hard to see that a node u is a descendant of a node v if and only if $\ell(v) \leq \ell(u) \leq \ell(f_v)$. Each node v stores the ranges $(\ell(u), \ell(f_u))$ for every child u of v. The label of a node v is the index $\ell(v)$. The routing process at a node v is performed as follows. If the message did not reach its destination t (namely, if $\ell(v) \neq \ell(t)$) then the node v checks if $\ell(v) \leq \ell(t) \leq \ell(f_v)$. If $\ell(v) \leq \ell(t) \leq \ell(f_v)$ then the message is sent to the child u of v such that $\ell(u) \leq \ell(t) \leq \ell(f_u)$. Otherwise, the message is sent to the parent of v. Note that the routing table used at a node v is of size $O(\deg(v) \cdot \log n)$-bits. While more efficient routing schemes on trees are known (e.g. [19]), it is unclear how to use them to route to a node v using only its index $\ell(v)$ which is needed is our scheme. Moreover, our scheme uses routing tables of size $\Omega(\deg(v) \cdot \log n)$, therefore this does not increase the asymptotic of the size of the routing tables.

Let us start with a high level overview of the way our routing scheme operates. Consider vertices $s, t \in V$ and suppose that a message is to be routed from s to t. Our routing process involves at most $\lceil \log(nW) \rceil$ iterations, where iteration i is expected to succeed in passing the message in the case where $2^{i-1} < \mathbf{dist}(s, t, G \setminus F) \leq 2^i$. As iteration i handles the possibility that $\mathbf{dist}(s, t, G \setminus F)$ is at most 2^i, it may ignore edges of weight greater than 2^i. Formally, let H_i be the set of G edges of weight greater than 2^i and let G_i be $G \setminus H_i$. Clearly, any two vertices that are connected in G by a path of length at most 2^i are still connected in G_i by the same path. Hence the routing process may be restricted to G_i. To facilitate each iteration i, in the preprocessing phase construct a tree cover $\mathbf{TC}_i = \mathbf{TC}(G_i, \omega, 2^i, k)$. Now for each tree $T \in \mathbf{TC}_i$ invoke our scheme for routing on trees with failures presented in Section 3 on the tree T and the graph G_i to assign each node $v \in T$ a label $L(v, T)$ and a routing table $A_v(T)$.

Each node v stores a routing table A_v, containing all the routing tables $A_v(T)$ together with some unique identifier $id(T)$ for each tree T such that $v \in T$ and $T \in \mathbf{TC}_i$ for some $1 \leq i \leq \lceil \log(nW) \rceil$.

In addition, for each node $t \in V$, let $T_i(t) \in \mathbf{TC}_i$ be the tree containing $B_{2^i}(t)$. The label $\mathcal{L}(t)$ of each node $t \in V$ has to store enough information about each $T_i(t)$ in order to allow routing on the tree $T_i(t)$ to the target t. Specifically, the label $\mathcal{L}(t)$ stores $L(t, T)$ together with the unique identifier $id(T)$ for the tree $T = T_i(t)$, for every $1 \leq i \leq \lceil \log(nW) \rceil$.

The routing process is schematically done as follows. Let F denote the set of failed edges at a given moment. In each iteration i from 1 to $\lceil \log(nW) \rceil$,

an attempt is made to route the message from the source s to the target t in the graph $G_i|_{T_i(t)} \setminus F$ using the tree $T_i(t)$ augmented with some additional information to be specified later on. If the routing is unsuccessful, i.e., it is not possible to route to t in $G_i|_{T_i(t)} \setminus F$, then s is informed by the routing scheme that it must proceed to the next iteration. Note that in order to route from s to t in the tree $T_i(t)$, the node s has to be familiar with the label $L(t, T)$ given to t by our scheme for the tree $T = T_i(t)$; this information can be extracted from t's label $\mathcal{L}(t)$.

In order to complete the description of our routing scheme, it remains to present our routing process from s to t in the graph $G_i|_{T_i(t)} \setminus F$. In what follows, we focus on describing our routing scheme in $H|_T \setminus F$ for a given graph H and a tree T, and describe the information stored in both the preprocessing phase and the routing phase.

3. Routing on a Tree with Faults

In this section we consider a given graph H and a tree T in H, and design a routing scheme such that when a set of edges F fails, if s and t are still connected in $H|_T \setminus F$, then our routing procedure manages to deliver a message from s to t on a path of length proportional to the depth of T, to F^3 and to some poly-log factor. More specifically, we prove the following lemma.

Lemma 3.1. *Consider a graph H with maximum edge weight W_H and a tree T of H ($E(T) \subseteq E(H)$). There is an efficiently constructible routing scheme such that given a source node s and a target node t, in the presence of a set of failures F (unknown to s), if s and t are connected in $H|_T \setminus F$, will deliver a message from s to t on a path of length $O(f^2(f \cdot diam(T) + f \cdot W_H + \log^2 n \cdot diam(T)))$, where $f = |F|$ and $diam(T)$ is the diameter of T. The size of the routing labels used by the scheme is $O(\log n)$ bits and the routing table stored at a node v is of size $O(\log^2 n \cdot \deg_H(v))$ bits, where $\deg_H(v)$ is the degree of v in the graph H.*

Observe that removing f edges from the tree T partitions it into $f + 1$ connected subtrees. The general goal of our routing scheme is to find a path from the subtree containing the source s to the subtree containing the target node t. This is done by searching for edges that reconnects these subtrees. Each time a new edge is discovered, the set of known subtrees that are reachable from s is increased. This process continues until either reaching the subtree containing t or until no such new edge can be found. In the latter the nodes can determine that t is not reachable from s in $H|_T \setminus F$.

The routing scheme described in this section is strongly based on the result of Duan and Pettie [11] on connectivity oracles with edge failures. We first review that result, and later show how to implement it in a distributed setting to achieve our fault-tolerant routing scheme. Their algorithm operates on a given spanning tree T of G. The algorithm traverses the tree T according to some Euler tour and constructs the list $L(T)$ of the vertices of $V(G)$ in the order in which they were encountered during this Euler tour, keeping only the first occurrence of

each vertex. For every v, let $\ell(v)$ denote v's index in $L(T)$. The algorithm then considers the adjacency matrix M of G, according to the ordering $L(T)$, and constructs a *range reporting* data structure on M (concretely, using the range reporting data structure of Alstrup et al. [2]). Duan and Pettie [11] observe that removing f edges from T partitions it into $f+1$ connected subtrees and splits $L(T)$ into at most $2f+1$ intervals, where the vertices of each connected subtree are the union of some subset of these intervals (see Figure 1 for illustration). Hence in order to decide connectivity in $G \setminus F$, it is enough to determine, for all pairs of the $2f+1$ intervals, if there is an edge in $E \setminus F$ connecting them. This can be done efficiently by the range reporting data-structure constructed on M.

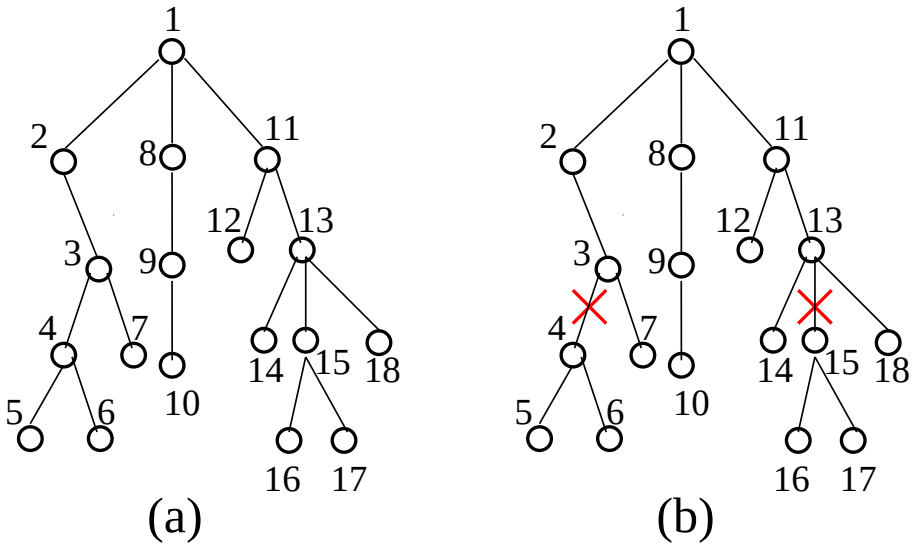

Figure 1: (a) The tree T and $L(T)$. (b) The edges $(3,4)$ and $(13,15)$ fail, resulting with the set of intervals $\{1-3, 4-6, 7-14, 15-17, 18\}$.

For the purpose of implementing the range reporting data structure in a distributed manner, we use a less efficient range reporting data structure. We first describe the well-known centralized range reporting data structure that is used in our construction, and later, in Subsections 3.1 and 3.2, we show how to implement this data structure in a distributed setting (which may be of independent interest).

Consider a boolean matrix $\tilde{M}$, in which queries of the following form need to be answered: "Given a range of rows (r_1, r_2) and a range of columns (c_1, c_2), return all cells that contain 1 in the matrix $\tilde{M}$ and are in the range of rows (r_1, r_2) and the range of columns (c_1, c_2)." The problem can be transformed to the following *planar range reporting* problem. Given a set P of N points in the plane, construct a data structure that can answer queries of the form: "return all points in the rectangle $[x_1, x_2] \times [y_1, y_2]$". This can be done by creating

a set of points $P(\tilde{M})$ containing a point $p(c)$ for each cell c in the matrix $\tilde{M}$ that contains 1 and setting its x-coordinate to be the row of the cell c and its y-coordinate to be the column of c. Now a query of the form $(r_1, r_2) \times (c_1, c_2)$ on $\tilde{M}$ can be answered by finding all points $P(\tilde{M})$ inside the rectangle $[r_1, r_2] \times [c_1, c_2]$. A data structure that can answer efficiently planar range reporting queries can be constructed as follows (cf. [10]). Given a set P of N nodes in the plane, first construct a main balanced binary tree $\mathcal{M}$ built on the x-coordinate of the points in P. For any internal or leaf node v in $\mathcal{M}$, let $P(v)$ be the subset of points of P corresponding to the subtree of v in $\mathcal{M}$. For any internal or leaf node v in $\mathcal{M}$, store a balanced binary tree $T(v)$ built on the y-coordinate of the points in $P(v)$. In addition, form a connected list chaining the leaves of $T(v)$, namely, provide each leaf u with a pointer to the next leaf with the lowest y-coordinate that is greater equal to the y-coordinate of u. Now given a query $[x_1, x_2] \times [y_1, y_2]$, the query algorithm first selects $O(\log n)$ canonical subsets that together contain all the points whose x-coordinate lie in $[x_1, x_2]$. Each such subset corresponds to a node in $\mathcal{M}$. In each such internal node $v \in \mathcal{M}$, the query algorithm searches for all nodes whose y-coordinate lie in $[y_1, y_2]$ using the search tree $T(v)$. The search on $T(v)$ takes $O(\log n + k')$ time, where k' is the number of nodes that lie in $[y_1, y_2]$ in $T(v)$. All in all, the query time is $O(\log^2 n + k)$, where k is the number of nodes reported. Note that if we want to report only $k'' \leq k$ points, then the query can be answered in $O(\log^2 n + k'')$ time.

Next, we turn to describe how to implement this range reporting data structure on M in a distributed setting. In Subsection 3.1 we describe the preprocessing phase, namely, the data that needs to be stored at the nodes in the preprocessing phase, Subsection 3.2 describes the routing phase and finally in Subsection 3.3 we prove the correctness of Lemma 3.1.

3.1. Preprocessing

In the preprocessing phase, construct the main balanced binary tree $\mathcal{M}$ built on the nodes V according to their order in $L(T)$, i.e., their indices $\ell(v)$. For an internal node $\tilde{v} \in \mathcal{M}$, let $P(\tilde{v})$ be the set of nodes in V corresponding to the subtree of $\tilde{v}$ in $\mathcal{M}$. For a set of nodes S, denote by $E_{out}(S)$ the set of edges in $E \setminus E(T)$ with exactly one endpoint in S. For an edge $e = (u, v) \in E_{out}(S)$, where w.l.o.g $u \in S$ and $v \notin S$, denote the incident node to e that is not in S by $Out(e, S) = v$. For each internal node $\tilde{v} \in \mathcal{M}$, construct a balanced binary tree $T(\tilde{v})$ on $E_{out}(P(\tilde{v}))$ sorted by $Out(e, P(\tilde{v}))$ according to their order in $L(T)$, namely, $T(\tilde{v})$ is a balanced binary tree whose nodes set is the set of edges $E_{out}(P(\tilde{v}))$. See Figure 2 for illustration.

Our algorithm looks for ways of progressing from its currently familiar "piece" of the network represented as an interval $I = (v_i, ..., v_j)$ of the ordered list $L(T)$ as explained earlier, to the "piece" containing the destination, which is another such interval I', disconnected from I by the faults of F. Note that in order to find all edges connecting some interval $I = (v_i, ..., v_j)$ with some other interval I', we need to check a set $X(I, I')$ of $O(\log n)$ internal nodes in $\mathcal{M}$. Consider some node $u \in V$, and let p be the path from the leaf representing u in $\mathcal{M}$ to the root of $\mathcal{M}$, each non-leaf node x on the path p has two children,

one that is part of the path p and another node y, if y is a left child, add it to $S_L(u)$, otherwise to $S_R(u)$. The node u itself is added to both $S_L(u)$ and $S_R(u)$. Note that the set $X(I, I')$ is a subset of $S_R(v_i) \cup S_L(v_j)$. Every node u stores an identifier of the internal nodes in $S_R(u) \cup S_L(u)$. As explained above, for each node $\tilde{v}$ in $S_R(u) \cup S_L(u)$, a balanced search tree $T(\tilde{v})$ is constructed, and the identifier of $\tilde{v}$ will contain an indication of the edge represented by the root of $T(\tilde{v})$. In addition, each internal node w in $T(\tilde{v})$ represents an edge $e = (x, y)$, where one node, say x, is inside $P(v)$ and the other, y, is outside it. The routing table of x stores the edges of the left and right children of w in $T(\tilde{v})$ as well as the ranges they represent, where by storing an edge (x, y) we mean storing $(L(x, T), L(y, T))$.

To summarize, the routing table $A_v(T)$ contains the identifiers of the internal nodes $S_R(v) \cup S_L(v)$, where the identifier of each internal node $\tilde{u} \in S_R(v) \cup S_L(v)$ contains the edge represented by the root of $T(\tilde{u})$. In addition, for every internal node $\tilde{u} \in \mathcal{M}$ such that $v \in P(\tilde{u})$ and every edge $(v, y) \in E_{out}(P(\tilde{u}))$, let w be the internal node in $T(\tilde{u})$ that represents (v, y), the routing table $A_v(T)$ stores the edges of the left and right children of w in $T(\tilde{u})$ as well as the ranges they represent (together with some identifier of the tree $T(\tilde{u})$). The routing table $A_v(T)$ of v also stores the index $\ell(v)$. The label $L(v, T)$ is just the index $\ell(v)$.

3.2. The Routing Process

In this section we show how to route a message from a source s to a target t on a single tree T with faults, allowing it to bypass the failures. After the failures of at most f edges, the tree is divided into at most $2f + 1$ intervals, and the goal is to reconnect these intervals, if possible. In the beginning of the routing process, the failed edges are not known to the nodes (except for their endpoints), so clearly, the different intervals are not known, and of course the way to reconnect the intervals is not known. During the routing process, some of this data is revealed and is attached to the header of the message. More specifically, the message accumulates information on the set of known intervals and the discovered edges that reconnect these intervals. Let $\mathcal{I}$ be the graph obtained by representing each discovered interval as a node and connecting two nodes if the process has already found an edge connecting the intervals they represent. The message carries along with it a copy of the graph $\mathcal{I}$. The goal is to reach as many intervals as possible in order to eventually reach t. As the nodes do not necessarily know all the failures, some of the intervals are "fake", in the sense that inside an interval there may be a failure (which the message still does not know about) that splits this interval, possibly into 3 intervals, where two of them are already connected. The intuition is that as long as the message does not encounter this failure, we do not care that the data it stores is imprecise, namely, if the message never encounters this failure, it means that this failure does not disturb our routing process and therefore we do not care about it. The other possibility is that eventually the message will encounter this failure, which will force it to update the set of intervals it carries along. Notice that the latter can happen at most f times, where f is the number of failed edges.

We now describe the routing process more formally. For simplicity, we first describe a solution where the message forwarded during the routing process is of size $O(|F|^2 \cdot \log n)$ bits, we later describe the small modifications needed to reduce the message size to $O(|F| \cdot \log n)$ bits. The header of the message contains the following additional data: it stores the set of currently known intervals, and for each pair of intervals I, I' it stores one of the following three items: a discovered edge $e(I, I')$ connecting them, an indication $discon(I, I')$ that these two intervals can not be connected, or an indication $Unknown(I, I')$ that it is still unknown if these two intervals can be connected. (Notice that some of the intervals are already connected by the original tree T.) The goal is to explore the pairs of intervals that are not decided yet, in order to eventually reach t if possible, or to conclude that t is not reachable from s in $H|_T \setminus F$.

At the beginning of this process, the source s is unaware of the failures, hence the set of intervals contains only one interval (which represents the entire tree T), and it just tries to route on T as if no failures occurred. The simplest scenario is that the path connecting s and t in T is free from failures and the message arrives at its destination. The more interesting case is when the routing process encounters some failure along the way and thus can not complete the normal routing process successfully.

In case a new failure is detected, the set of intervals is updated accordingly, and the set of *recovery edges* is updated as well, where by a recovery edge we mean an edge that reconnects two intervals that are not connected in $T \setminus F$. Namely, assume the interval I is now split into at most three intervals I_1, I_2, I_3, and consider an edge $e(I, I')$ that was previously believed to connect I to some other interval I'. This edge now connects one of the three intervals I_1, I_2, I_3 to I', and as we know the incident nodes of the edge, there is no problem to identify the right one, say, I_1, and update the information in the message header to include $e(I_1, I')$. In addition, if it is already known that there is no way to reconnect I to some other interval I' (i.e., the message header contains $discon(I, I')$), then it is also impossible to reconnect I' to any of I_1, I_2, I_3 and we update the data in the message header by storing in it the indicators $discon(I_1, I')$, $discon(I_2, I')$ and $discon(I_3, I')$.

At any stage during the routing process, the routing process may be in one of three states. The first state is that the nodes can check if using the recovery edges collected so far it is possible to reach t (for example, by running a DFS on the graph $\mathcal{I}$). In the second state, the nodes can verify if it is impossible to reach t in $H|_T \setminus F$. This case happens when it is already known that all the intervals that are reachable from s can not be connected to any other interval and t is still not reachable from s, where we say that an interval I is reachable from some node $v \in T$ if the interval containing v and the interval I are connected in $\mathcal{I}$. The third state is where it is still unclear if it is possible to reach t in $H|_T \setminus F$.

In the first state, the message is sent to t (assuming no new failures are detected during the remaining part of the journey). In the second state, an indication that t is not reachable in $H|_T \setminus F$ is sent to s. In the third state, pick a pair of intervals (I, I') that have not been decided yet (i.e., such that

the message header contains $Unknown(I, I')$) and such that I is reachable from the current node (and therefore also from s). If no such pair exists, it can be determined that the routing is not possible. Next, search for a recovery edge connecting I and I' as follows. First, forward the message to some node in I. Once reaching I, check the potential search balanced trees, namely, the search trees for the interval I. This data can be extracted from the routing tables of the first and last nodes v_i and v_j in the interval I, recall that the set the potential search trees is a subset of $S_R(v_i) \cup S_L(v_j)$. (Note that in order to route to the nodes v_i and v_j, the nodes only need to be familiar with the indices $\ell(v_i)$ and $\ell(v_j)$.) Now for each potential search tree T_1 on an interval $I_1 \subseteq I$, search for a recovery edge reconnecting to I'. The search on T_1 can be done as follows. First reach the node z containing the root of T_1 (recall that the root of T_1 represents an edge (x, y) where x is in I_1 and y is outside, so $z = x$ and that the edge (x, y) is part of the identifier of T_1). The node z stores in its routing table the left and right children of the root of T_1 as well as the ranges they represent, so the node z knows if it supposed to continue the search on the left or right child. This process continues until reaching the leaves. If the leaf is not in the range I', then it's not hard to see that I_1 and I' can not be connected. Otherwise, there are two subcases. If the edge in the leaf is not faulty, then we found a recovery edge reconnecting I and I'. The second subcase is that this edge is faulty. Recall that each leaf stores an identifier to the next leaf, so we can just check the next leaf, and so on. This is repeated for all $O(\log n)$ potential subintervals of I, until either finding the desired recovery edge connecting I and I' or deciding that it is impossible to connect I to I'.

3.3. Analysis

This section is devoted to proving Lemma 3.1.

Correctness.. We need to prove that if s and t are connected in $H|_T \setminus F$, then t will get the message. This follows almost trivially from [11] and is proved in the following lemma.

Lemma 3.2. *If s and t are connected in $H|_T \setminus F$, then the message will reach its destination using our routing scheme.*

Proof: Assume s and t are connected in $H|_T \setminus F$. Consider only final intervals, namely, intervals that were not split any more during the routing process. Let $\mathcal{I}_{fin}$ be the final graph $\mathcal{I}$. It's not hard to verify that there must be a path of final intervals in $\mathcal{I}_{fin}$ connecting the final interval of s with the final interval of t. As our algorithm checks all possible pairs of intervals, it will eventually find that path (or some other path p reaching t). Therefore, either the algorithm manages to reach t on p, or it detects another failure on p, but this is in contradiction with the fact that we consider final intervals. ∎

Stretch Analysis..

Lemma 3.3. *If s and t are connected in $H|_T \setminus F$, then the length of the path obtained by our routing scheme on $H|_T \setminus F$ is at most $O(f^2(f \cdot diam(T) + f \cdot W_H + \log^2 n \cdot diam(T)))$.*

Proof: As mentioned above, there are at most $O(f)$ real intervals. Therefore, at most $O(f^2)$ pairs of real intervals need to be examined. In the beginning, the routing process is unaware of these intervals, so it might check "fake" intervals. At first glance, this appears to be a waste and it seems that we might need to check more than $O(f^2)$ pairs of intervals. But a more careful inspection reveals that it is enough to check only $O(f^2)$ pairs of intervals. To see why this is true, assume we have already checked the pair of intervals I and I' and at some point discover that I splits into two or three intervals, I_1, I_2, I_3. There are two cases. The first is that we discovered that I and I' can not be connected. In that case we can determine that all I_1, I_2, I_3 can not be connected to I', and actually this works to our advantage, as in one check we saved two or three checks. The second case is where I and I' were connected by an edge $e = (u, v)$ such that $u \in I$ and $v \in I'$. Notice that u must belongs to one of I_1, I_2, I_3. Assume w.l.o.g. that it belongs to I_1. Then we can determine that I_1 and I' are connected by the edge e. In addition, each time we try to discover if two intervals are connected or not, we either determine if this pair can be connected or not, or we discover another failure. As mentioned earlier the latter can happen at most f times. To conclude, there are at most $O(f^2)$ such checks.

Next we bound the maximum length of the path obtained by a single check of two intervals. Assume we want to check the pair of intervals I and I'. It must be the case that one of the intervals is reachable by the edges of T together with the recovery edges discovered so far. Therefore, the path leading to the relevant interval is of length at most $(f + 1) \cdot diam(T) + f \cdot W_H$, where $diam(T)$ is the diameter of T. To see this, note that the diameter of each of the connected subtrees of $T \setminus F$ is at most $diam(T)$, and in addition, the path uses at most f recovery edges of weight at most W_H each. Next, we bound the length of the subpath used for checking the pair of intervals I and $I' = [a, b]$ once we are already in I. It's enough to bound the maximum length assuming no new faulty edge was encountered during the way. We know the interval I, and in particular we know its first and last nodes and thus can easily find the id's of the $O(\log n)$ search trees that need to be examined. We now bound the length of one of these search trees T_1. The id of the tree T_1 contains the root node of the tree T_1 and each intermediate node contains the id's of its left and right children and the ranges they represent. Moreover, we assume that the interval is connected, so every two nodes in the interval are reachable using only the edges of T. We start with the root of T_1 and search for the edge e of minimum $\ell(Out(e, S))$ that is equal to or greater than a, where the set S is the set of nodes $P(r(T_1))$ where $r(T_1)$ is the root of T_1. Checking the ranges of children of the root, we know if we need to move left or right, and we continue in this way until reaching the relevant leaf. Note that a child and a parent in the tree T_1 are not necessarily adjacent in T, but their distance is at most $diam(T)$. The depth of T_1 is at most $\log n$, and for each step we might need to travel a distance

at most $diam(T)$. Therefore the total length for reaching the relevant leaf for a single search tree is at most $\log n \cdot diam(T)$. Summing over all search trees, the total length is $O(\log^2 n \cdot diam(T))$. Note that once reaching a leaf in one of the search trees, this leaf might represent a faulty edge, therefore we might need to check the next leaf and so on, until either reaching a leaf that is not in I' or finding a non-faulty edge in I'. Notice that this can happen at most f times for all search trees (since the sets of edges of the search trees are disjoint), and in addition, moving from one leaf to the next can be done along a path of length at most $diam(T)$. All in all, the total length of the path traversed on T is $O(f^2(f \cdot diam(T) + f \cdot W_H + \log^2 n \cdot diam(T)))$. ▮

The label of a node v is just the index $\ell(v)$ and thus of size $O(\log n)$ bits. In addition, v participated in $O(\log n)$ search trees, and for each such search tree, $O(\log n \cdot \deg_H(v))$ bits are stored in the routing table $A_v(T)$ of v. We get that the routing table $A_v(T)$ is of size $O(\log^2 n \cdot \deg_H(v))$ bits.

Together with Lemmas 3.2 and 3.3, Lemma 3.1 follows.

4. Analysis for the entire Routing Scheme

In this section we analyze our routing scheme. Let p be a shortest path connecting s and t in $G \setminus F$, and let i be the index such that $2^{i-1} \leq \mathbf{dist}(P) = \mathbf{dist}(s, t, G \setminus F) \leq 2^i$.

The following lemmas prove Theorem 1.1.

Lemma 4.1. *The nodes s and t are connected in $G_i|_{T_i(t)} \setminus F$.*

Proof: First note that all edges in p are of weight at most 2^i and thus exist in G_i, and moreover as p is a path in $G \setminus F$, all these edges are non-faulty and thus occur also in $G_i \setminus F$. Moreover, $T_i(t)$ contains all nodes at distance at most 2^i from t. We get that all nodes in p are in $T_i(t)$. The lemma follows. ▮

Lemma 4.2. *The path obtained by our routing scheme is of length at most $O(f^2 \cdot k \cdot \mathbf{dist}(s, t, G \setminus F)(f + \log^2 n))$.*

Proof: By Lemma 4.1, our routing scheme performs at most i iterations. By Lemma 3.3, the path traversed in each iteration j is of length at most $O(f^2(f \cdot 4(2k-1)2^j + f \cdot W_{T_j(t)} + \log^2 n \cdot 4(2k-1)2^j)) = O(f^2(f \cdot k \cdot 2^j + f \cdot 2^j + \log^2 n \cdot k \cdot 2^j))$. Summing over all iterations, the lemma follows. ▮

Lemma 4.3. *The size of the routing table of a node v is at most $\lceil \log(nW) \rceil \cdot k \cdot n^{1/k} \cdot \deg(v) \cdot \log^2 n$ bits.*

Proof: There are $\lceil \log(nW) \rceil$ iterations. For each iteration i, v belongs to at most $k \cdot n^{1/k}$ trees in $\mathbf{TC}_i$. For each such tree T the node v stores in its routing table $A_v(T)$. By Lemma 3.1, $A_v(T)$ is of size $O(\log^2 n \cdot \deg_G(v))$ bits. The lemma follows. ▮

Lemma 4.4. *The labels are of size* $\lceil \log(nW) \rceil \cdot \log n$ *bits.*

Proof: Consider a node $t \in V$. The label of t is the concatenation of the labels given to t by our routing scheme for trees with failures for each tree $T_i(t)$ for every $1 \leq i \leq \lceil \log(nW) \rceil$. By Lemma 3.1, the size of each such label is $O(\log n)$ bits, and thus the label of t is of size $\lceil \log(nW) \rceil \cdot \log n$ bits. ▌

Lemma 4.5. *The message forwarded during the routing process is of size* $O(|F|^2 \cdot \log n)$ *bits.*

Proof: It's not hard to see that $O(\log n)$ bits are stored in the message for each pair of intervals . ▌

In fact, the message size can be reduced to $O(|F| \cdot \log n)$ bits. To see this, note that in order to decide connectivity in $G_i|_{T_i(t)}$, there is no need to store information on all pairs of intervals; rather, it's enough to store at most $|F|$ recovery edges and an indication on which pairs of intervals have already been examined. This can be done using $O(|F| \cdot \log n)$ bits as follows. Store four lists of intervals: ReachOldList, ReachNewList, NonReachList and InCheckList. The first two lists contain intervals that are already known to be reachable from s, the first list represents "old" intervals and the second "new" discovered intervals. The list NonReachList represents intervals that are not reachable from s or from any interval in the list ReachOldList. The list InCheckList represents intervals whose reachability has not yet been checked. In the beginning all lists but ReachOldList are empty and the list ReachOldList contains the interval representing the entire tree. In each iteration the first interval I in the list InCheckList is checked against the intervals of the list ReachOldList (one by one) until either finding a recovery edge and deciding that I is reachable from s or deciding that it is not reachable from any interval in the list ReachOldList. In the first case, I is added to the list ReachNewList, and in the second case it is added to the list NonReachList. If during the routing process a faulty edge is encountered on one of the reachable intervals I, then the interval I splits into two or three intervals, I_1, I_2, I_3. One or two of these intervals are now not reachable from s, so we add these intervals back to the list InCheckList. In addition, substitute the interval I by the intervals from the set $\{I_1, I_2, I_3\}$ that are reachable from s. When the list InCheckList becomes empty, move all intervals from the list NonReachList to the list InCheckList and set the list ReachOldList to be the list ReachNewList.

We note that if we care only for the total size of the routing tables at all the nodes, then it's possible to reduce the stretch bound by a logarithmic factor. This can be achieved as follows. Recall that the algorithm stores for each internal node v in $\mathcal{M}$ a balanced search tree $T(v)$ in a distributed manner. Let I be the interval of nodes corresponds to the internal node v. Instead of storing the tree $T(v)$ distributively on all nodes in I, the algorithm can just pick an arbitrary node x in I and store the entire tree $T(v)$ in x. It's not hard to see that this will reduce the stretch by a logarithmic factor.

Acknowledgement: I would like to thank my advisor, David Peleg, for very helpful ideas, comments and observations. I also thank Michael Langberg and Liam Roditty for useful discussions.

References

[1] I. Abraham, S. Chechik, C. Gavoille and D. Peleg. Forbidden-set distance labels for graphs of bounded doubling dimension. In PODC, pages 192–200, 2010.

[2] S. Alstrup, G. S. Brodal and T. Rauhe. New Data Structures for Orthogonal Range Searching. *Proc. 41st IEEE Symp. on Foundations of Computer Science (FOCS)*, 198–207, 2001.

[3] B. Awerbuch, A. Bar-Noy, N. Linial, and D. Peleg. Improved routing strategies with succinct tables. *J. Algorithms*, 307–341, 1990.

[4] B. Awerbuch, S. Kutten, and D. Peleg. On buffer-economical store-and-forward deadlock prevention. In *Proc. INFOCOM*, pages 410–414, 1991.

[5] B. Awerbuch and D. Peleg. Sparse partitions. *31st FOCS*, 503–513, 1990.

[6] S. Chechik, M. Langberg, D. Peleg, and L. Roditty. f-sensitivity distance oracles and routing schemes. *18th ESA*,84–96, 2010.

[7] E. Cohen. Fast algorithms for constructing t-spanners and paths with stretch t. In *Proc. IEEE Symp. on Foundations of Computer Science*, pages 648–658, 1993.

[8] B. Courcelle and A. Twigg. Compact forbidden-set routing. *STACS*, 37–48, 2007.

[9] L.J. Cowen. Compact routing with minimum stretch. *J. Alg.*, 38:170–183, 2001.

[10] M. de Berg, O. Cheong, M. van Kreveld and M. Overmars. Computational Geometry: Algorithms and Applications. *Springer*, 2008.

[11] R. Duan and S. Pettie. Connectivity oracles for failure prone graphs. In *Proc. ACM STOC*, 2010.

[12] T. Eilam, C. Gavoille, and D. Peleg. Compact routing schemes with low stretch factor. *J. Algorithms*, 46:97–114, 2003.

[13] P. Fraigniaud and C. Gavoille. Memory requirement for universal routing schemes. *14th PODC*, 223–230, 1995.

[14] C. Gavoille and M. Gengler. Space-efficiency for routing schemes of stretch factor three. *J. Parallel Distrib. Comput.*, 61:679–687, 2001.

[15] C. Gavoille and D. Peleg. Compact and localized distributed data structures. *Distributed Computing*, 16:111–120, 2003.

[16] D. Peleg. *Distributed computing: a locality-sensitive approach*. SIAM, Philadelphia, PA, 2000.

[17] D. Peleg and E. Upfal. A trade-off between space and efficiency for routing tables. *J. ACM*, 36(3):510–530, 1989.

[18] N. Santoro and R. Khatib. Labelling and implicit routing in networks. *The Computer Journal*, 28(1):58, 1985.

[19] M. Thorup and U. Zwick. Compact routing schemes. In *Proc. 13th ACM Symp. on Parallel algorithms and architectures (SPAA)*, 1–10, 2001.

[20] M. Thorup and U. Zwick. Approximate distance oracles. *J. ACM*, 52, 1–24, 2005.

[21] D. A. Twigg. *Forbidden-set Routing.* PhD thesis, University of Cambridge (King's College), 2006.

[22] J. van Leeuwen and R. Tan. Computer networks with compact routing tables. *G. Rozemberg and A. Salomaa, editors, The book of L*, 259-273, 1986.

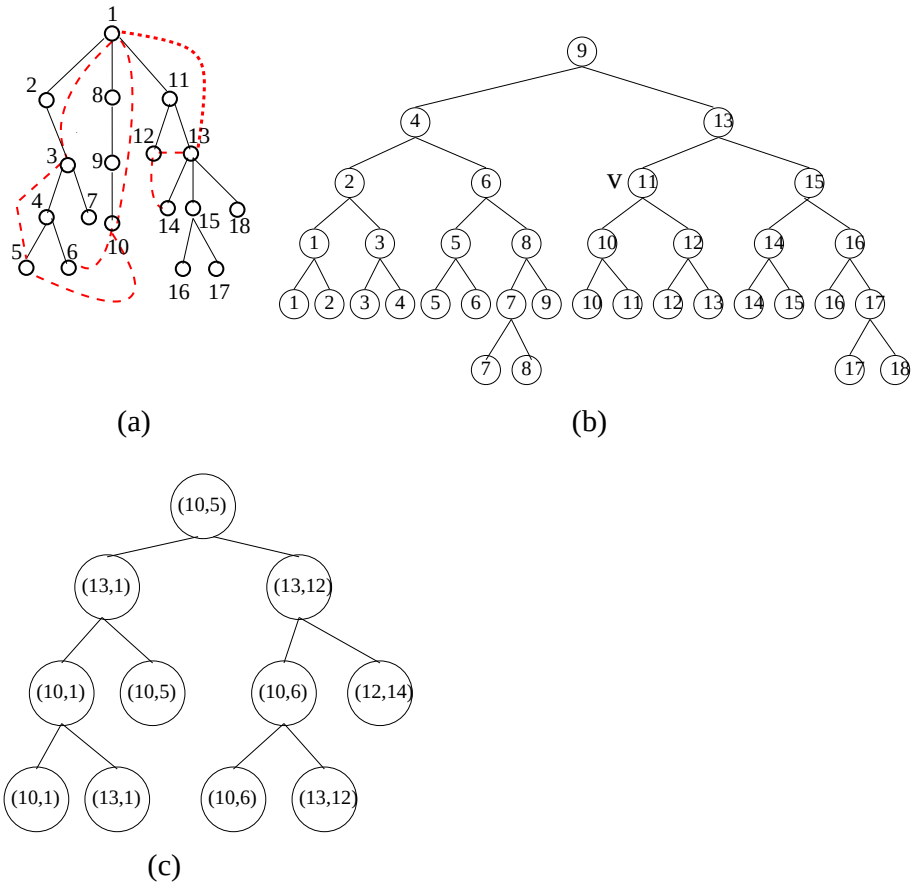

Figure 2: (a) The tree T and $L(T)$. Black solid edges are the tree edges and the red dashed edges are the remaining edges in the induced graph. (b) The main balanced binary tree $\mathcal{M}$. (c) The balanced binary tree of the inner node v in $\mathcal{M}$, representing the interval $10-13$. Note that the set of edges $E_{out}(P(v))$ satisfies $E_{out}(P(v)) = \{(10,1),(13,1),(10,5),(10,6),(13,12),(12,14)\}$, where the edges are sorted by $Out(e,P(v))$.